Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Evoluções na Biblioteca de Transformações RJTL

Uriel de Barcelos Conceição Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Orientador
Prof. Dr. Rodrigo Bonifácio

Brasília
2018

Universidade de Brasília — UnB
Instituto de Ciências Exatas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Coordenador: Prof. Dr. Edison Ishikawa

Banca examinadora composta por:

Prof. Dr. Rodrigo Bonifácio (Orientador) — CIC/UnB
Prof. Dr. Edna Dias Canedo — CIC/UnB
Prof. Dr. Edson Alves da Costa Júnior — FGA/UnB

CIP — Catalogação Internacional na Publicação

Silva, Uriel de Barcelos Conceição.

Evoluções na Biblioteca de Transformações RJTL / Uriel de Barcelos Conceição Silva. Brasília : UnB, 2018.
43 p. : il. ; 29,5 cm.

Monografia (Graduação) — Universidade de Brasília, Brasília, 2018.

1. Evolução de linguagens, 2. Refatoração, 3. Transformação de Programas.

CDU 004

Endereço: Universidade de Brasília
Campus Universitário Darcy Ribeiro — Asa Norte
CEP 70910-900
Brasília–DF — Brasil

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Evoluções na Biblioteca de Transformações RJTL

Uriel de Barcelos Conceição Silva

Monografia apresentada como requisito parcial
para conclusão do Bacharelado em Ciência da Computação

Prof. Dr. Rodrigo Bonifácio (Orientador)
CIC/UnB

Prof. Dr. Edna Dias Canedo
CIC/UnB

Prof. Dr. Edson Alves da Costa Júnior
FGA/UnB

Prof. Dr. Edison Ishikawa
Coordenador do Bacharelado em Ciência da Computação

Brasília, 09 de julho de 2018

Dedicatória

Dedico este trabalho a todos aqueles que me auxiliaram ao longo desses anos e contribuíram para a minha formação.

Agradecimentos

Gostaria de agradecer primeiramente à minha família, que tanto me apoiou durante a graduação. Ao professor Rodrigo, por ter me ajudado ao longo do curso e me proporcionado várias oportunidades de aprendizado. À Rebeca, que esteve comigo nos momentos difíceis e me apoiou nos momentos em que mais precisei, sempre acreditando em mim e me fazendo seguir em frente. Por fim, agradeço aos amigos que fiz na CJR, que tornaram esses 6 anos de graduação muito mais divertidos.

Resumo

Sistemas de software evoluem frequentemente ao longo do tempo, seja por novos requisitos de negócio ou necessidades técnicas. Linguagens de programação evoluem de uma maneira semelhante, tornando recursos e construções antigas obsoletas. A existência de ambos recursos novos e obsoletos de uma linguagem traz problemas como um maior custo da manutenção e maior curva de aprendizado para novos desenvolvedores. Para atacar este problema, foi implementada uma biblioteca denominada RJTL, que visa refatorar sistemas legados Java a fim de substituir construções obsoletas por construções introduzidas em versões mais recentes da linguagem Java. O presente trabalho se propõe a dar continuidade na implementação da RJTL, evoluindo limitações existentes e também apresentando novas transformações. Os resultados foram avaliados aplicando as transformações da ferreamenta em um conjunto de projetos Java, e apresentam uma pequena queda nos erros encontrados na implementação inicial e também a obtenção de novos cenários passíveis de transformação identificados pela ferramenta.

Palavras-chave: Evolução de linguagens, Refatoração, Transformação de Programas.

Abstract

Software systems evolve frequently over time, either due to new business requirements or technical needs. Programming languages evolve in a similar way, making old language constructs and resources obsolete while doing so. The existente of both new constructs as well as obsolete ones leads to some problems, such as a higher maintenance cost and a steeper learning curve for new developers. To solve this problem, a library called RJTL was implemented, aiming to refactor legacy Java systems in order to replace old language constructs for ones introduced in recent Java versions. This work aims to continue the implementation of RJTL, presenting evolutions to existing limitations as well as new transformations.

Keywords: Language Evolution, Refactoring, Program Transformation.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

De maneira semelhante a sistemas de software, linguagens de programação estão em constante evolução. À medida que novas demandas e requisitos surgem, sistemas de software precisam ser atualizados e evoluídos para atender tais necessidades. Analogamente, uma linguagem de programação precisa ser constantemente atualizada para satisfazer as necessidades de seus usuários. No entanto, esse processo de evolução traz alguns problemas (5), devido à necessidade de se manter a retrocompatibilidade com certos recursos de uma linguagem que podem se tornar obsoletos, uma vez que caso contrário, programas que utilizam desses recursos seriam forçados a se manterem em versões antigas, para que possam continuar funcionando. Ao mesmo tempo, a falta de novas atualizações de uma linguagem pode torná-la obsoleta e não atraente, à medida que outras linguagens e novas construções são adotadas pelos desenvolvedores.

Sendo assim, recursos de versões diferentes de uma linguagem acabam coexistindo dentro de sistemas que possuem uma longa existência. Isso faz com que a curva de aprendizado seja mais íngreme para novos desenvolvedores (5), que precisam aprender a utilizar construções de diferentes versões da linguagem, além de tornar a manutenção desses sistemas mais difícil.

A solução proposta por Overbey e Johnson (5) para lidar com este problema é a utilização de ferramentas automatizadas de refatoração a fim de substituir construções obsoletas por novos recursos introduzidos em versões posteriores. Isso faria com que os programas construídos estivessem sempre utilizando recursos atuais de uma linguagem, e eliminaria a preocupação dos projetistas de manter a retrocompatibilidade à medida de que novas versões são lançadas. Idealmente, toda vez que uma linguagem disponibilizasse uma nova versão, também seria disponibilizada uma ferramenta que atualizasse uma base de código para que esteja *up-to-date* em relação à nova versão.

Uma iniciativa para implementar essa ferramenta foi a *RJTL* (*Rascal Java Transformation Library*)(6), uma biblioteca criada para apoiar a evolução de código Java a fim de utilizar construções mais recentes da linguagem por meio de refatorações. Essa biblioteca oferece um conjunto de transformações que introduzem o recurso *multi-catch*, o *Diamond Operator*, ambos adicionadas na versão Java 7, e o uso de expressões lambda, presentes a partir da versão 8, e foi desenvolvida utilizando a linguagem Rascal (3), que será detalhada no capítulo 2.

A implementação atual da RJTL possui algumas limitações, principalmente no que se refere à verificação de pré-condições necessárias para aplicar algumas transformações, e

também oferece um conjunto restrito de transformações. Este trabalho tenta atacar essas limitações e seus objetivos serão descritos a seguir.

1.1 Objetivos

O objetivo geral deste trabalho consiste em evoluir a implementação atual da biblioteca RJTL. Esse objetivo se decompõe em três objetivos específicos:

- Redução na ocorrência de falsos positivos e negativos nas transformações já implementadas, realizando a verificação de pré condições necessárias para a alicação de algumas dessas transformações. Para alcançar esses objetivos, foi implementado um módulo escrito em Java que permite auxiliar nessa verificação.
- Extensão dos cenários identificados pelas transformações já implementadas. Em particular, a transformação *Diamond Operator* foi extendida para identificar um novo cenário.
- Construção de uma nova transformação, denominada *Method Reference*.

1.2 Metodologia

Para atingir os objetivos deste trabalho, foi realizada uma revisão da literatura, relacionada à refatoração de código, rejuvenescimento de software e ferramentas de análise e manipulação de código (Rascal e Java Parser).

Em seguida, foi feita a evolução da biblioteca RJTL, tanto com o objetivo de mitigar limitações da implementação existente quanto com o objetivo de implementar novas transformações.

Finalmente, foi conduzido um estudo empírico com o objetivo de verificar os resultados da implementação. Para isso, foram selecionados alguns projetos *open source* para se aplicar a RJTL contendo as evoluções implementadas.

1.3 Estrutura do Documento

Este documento está dividido em quatro capítulos, sendo o primeiro destes o presente capítulo (Introdução). No capítulo seguinte, Fundamentação Teórica, serão apresentados alguns conceitos importantes para a ambientação do leitor no contexto do trabalho, tais como Refatoração, uma breve descrição de algumas ferramentas utilizadas no domínio de Transformação de Programas, um detalhamento da linguagem Rascal, a evolução histórica da linguagem Java e uma explicação mais detalhada sobre o trabalho anterior de desenvolvimento da RJTL.

O Capítulo 3 descreve em detalhes a solução proposta para alcançar os objetivos do trabalho, que consiste no módulo Java que realiza a interface com o código Rascal já existente, bem como as mudanças nas transformações já existentes e o funcionamento da transformação *Method Reference*. Neste capítulo também serão descritos os resultados coletados na aplicação das transformações a projetos *open source*.

Por fim, no Capítulo 4 serão apresentadas as conclusões do estudo realizado e a possibilidade de trabalhos futuros que podem ser feitos a partir das contribuições contidas neste trabalho.

Capítulo 2

Fundamentação Teórica

Este capítulo visa apresentar os conceitos necessários para a contextualização do leitor em relação ao trabalho realizado. Esses conceitos incluem a Evolução da Linguagem Java, Refatoração e as ferramantas existentes que são utilizadas no contexto de Transformação de Programas.

2.1 Refatoração

De acordo com Fowler (7), Refatoração é o processo de alterar um sistema de *software* de maneira que sua estrutura interna seja melhorada, preservando seu comportamento externo. É uma maneira de evoluir código a fim de minimizar a possibilidade de adição de *bugs*, além de poder alterar o *design* de um determinado componente de código que já foi escrito anteriormente. Exemplos de refatorações incluem alterar nomes de classes ou métodos, mover métodos e campos de uma classe para outra ou ainda extrair um método a partir de um trecho de código.

É importante ressaltar que o conceito de refatoração é utilizado apenas quando o propósito das alterações em um programa é torná-lo mais fácil de ser entendido e modificado. Existem outros tipos de transformações de programas, como definido por Visser (8). Visser divide transformações de programas em dois tipos: *tradução*, quando a transformação resulta em uma mudança de linguagem de programação, e *reformulação*, quando as linguagens fonte e alvo são as mesmas.

Dentro das transformações caracterizadas como reformulações, Visser define ainda quatro subcategorias:

- *Normalização*: Processo em que certas construções mais complexas de uma linguagem são substituídas por expressões mais simples da mesma linguagem.
- *Otimização*: Busca alterar trechos de um programa a fim de otimizar o tempo de execução ou espaço em memória alocado pelo programa.
- *Renovação*: Busca modificar o comportamento de um trecho de código, visando corrigir algum erro ou atender uma nova necessidade.

A quarta e última subcategoria é a Refatoração, cuja definição já foi apresentada. Este trabalho foca no contexto de refatorações e normalizações, mais especificamente na

aplicação de transformações para evoluir programas *Java* utilizando novas construções da linguagem.

As transformacões aplicadas ao longo do trabalho foram construídas utilizando Rascal (3), uma Linguagem Especifica do Domínio (*DSL*) criada para resolver problemas no domínio de análise e manipulação de código. A Seção 2.2 justifica o uso da linguagem Rascal em relação a outras ferramentas existentes e detalha um pouco mais o seu funcionamento.

2.2 Ferramentas para Transformação de Programas

Esta subseção apresenta algumas alternativas existentes que são utilizadas no domínio de Análise e Manipulação de Código Fonte (*Source Code Analysis and Manipulation*), em particular as ferramentas StrategoXT, Spoofax e Rascal, e também justifica o uso do Rascal no contexto deste trabalho.

2.2.1 StrategoXT

Stratego/XT (1) é uma infraestrutura genérica construída para criar aplicações que realizam transformações de programas de maneira independente. Para isso, combina Stratego, uma linguagem destinada à implementação de transformações, com XT, um conjunto de componentes reutilizáveis e ferramentas criadas para o desenvolvimento de aplicações também destinadas à transformação de programas. Stratego/XT tem o propósito de ser utilizada na análise, manipulação e geração de programas, apesar de seus recursos poderem ser utilizados na transformação de outros tipos de documentos.

A Figura 2.1 apresenta a organização do Stratego/XT em camadas:

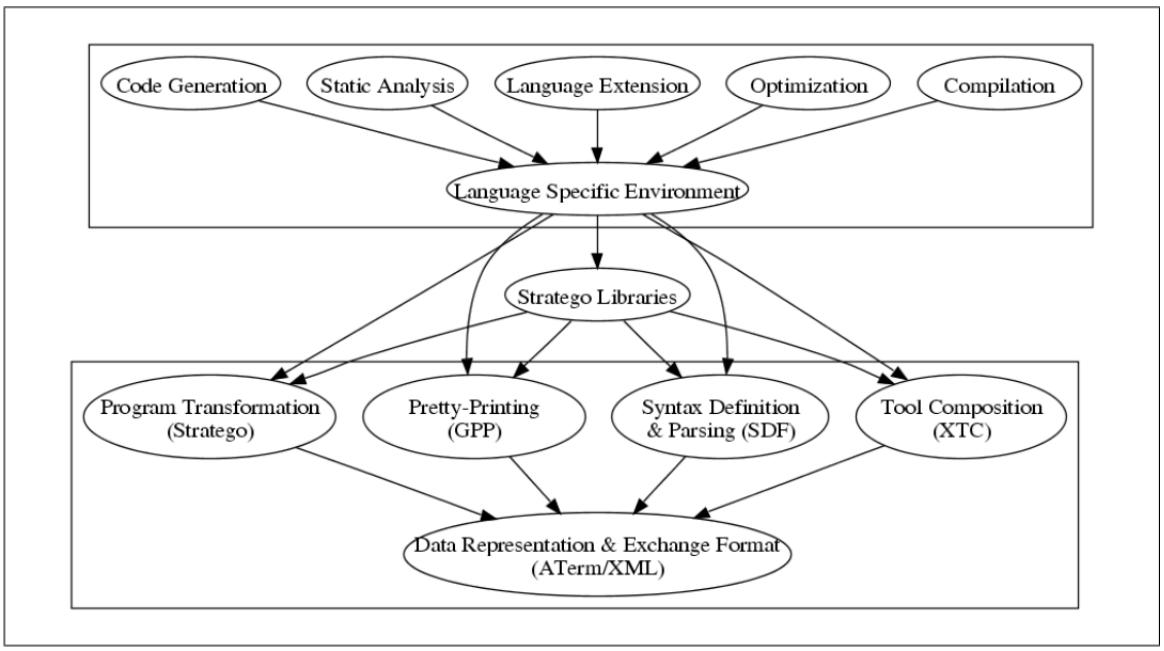


Figura 2.1: Ilustração da organização do StrategoXT em camadas (1).

A linguagem Stratego (2) fornece alguns recursos para a transformação de programas, tais como:

- **Regras de reescrita**: é a maneira de expressar transformações básicas de um programa, onde o padrão à esquerda $p1$ de uma regra do tipo $R : p1 \rightarrow p2$ é substituído pelo padrão à direita $p2$, caso uma determinada expressão de um programa seja associada a uma regra por meio de casamento de padrões.
- **Regras de reescrita dinâmicas**: são o mecanismo proposto para lidar com transformações sensíveis ao contexto, uma vez que regras de reescrita tradicionais lidam apenas com expressões livres de contexto.
- **Estratégias programáveis de reescrita**: Stratego permite a definição de algoritmos para a aplicação de regras de reescrita, a fim de evitar problemas como não-terminação ou possível diferença de resultados ao aplicar as regras em ordens diferentes. Regras podem ser combinadas de maneira sequencial e aplicadas em uma determinada expressão conforme a definição de uma estratégia.
- **Sintaxe concreta de objetos**: ao invés de representar um programa por meio de Árvores Sintáticas Abstratas (*Abstract Syntax Trees*), Stratego permite a utilização da sintaxe concreta (9) da linguagem utilizada pelo programa a ser transformado, a partir de uma definição formal da sintaxe, utilizando a definição formal SDF (10).

A Figura 2.2 apresenta um exemplo de código utilizando a linguagem Stratego:

```
module lambda-transform
imports lambda-sig lambda-vars iteration simple-traversal
rules
  Beta : App(Abs(x, e1), e2) -> <lsubs>([(x,e2)], e1)
strategies
  simplify = bottomup(try(Beta))
  eager = rec eval(try(App(eval, eval)); try(Beta; eval))
  whnf  = rec eval(try(App(eval, id)); try(Beta; eval))
```

Figura 2.2: Exemplo de código Stratego (2).

O conjunto de ferramentas XT complementa a linguagem Stratego dentro da infraestrutura Stratego/XT. XT possibilita a definição de sintaxes de linguagens de programação utilizando a definição formal SDF de uma maneira declarativa. O trecho de código a seguir ilustra uma definição de sintaxe dentro do XT:

```
module Expressions
exports
    sorts Id Exp
    lexical syntax
        [A-Za-z][A-Za-z0-9]* -> Id
    context-free syntax
        Id -> Exp {cons("Var")}
        Exp "+" Exp -> Exp {cons("Plus"),left}
        "if" Exp "then" Exp "else" Exp "end" -> Exp {cons("If")}
```

A partir dessa definição, XT fornece alguns componentes que podem ser úteis na construção de transformações, tais como a conversão de *ASTs* em um arquivo de código fonte (*Pretty-Printing*), geração de *parsers* ou a verificação da estrutura de uma *AST*. Esses componentes podem ser compostos sequencialmente, constituindo um *pipeline* que pode ser utilizado em conjunto com as estratégias de reescrita da linguagem *Stratego* definidas anteriormente.

A principal limitação da ferramenta *Stratego/XT* é a sua dificuldade de utilização, tanto para realizar a configuração do ambiente de desenvolvimento quanto o fato de sua sintaxe diferir consideravelmente das linguagens mais utilizadas pela comunidade.

2.2.2 Spoofax

Devido à alta barreira de entrada para se utilizar ferramentas e linguagens voltadas para transformação de programas, seja pelo fato da maioria dessas linguagens utilizarem paradigmas de programação que são menos familiares a novos usuários, ou pela falta de existirem ambientes modernos de desenvolvimento específicos (11), foi proposto *Spoofax*, um ambiente interativo de desenvolvimento baseado na plataforma *Eclipse*, com o objetivo de facilitar o desenvolvimento de sistemas de transformação de programas, utilizando Stratego/XT.

Spoofax fornece vários recursos que auxiliam o desenvolvimento, tais como *syntax highlighting*, um editor de texto para programas *Stratego* e definições de sintaxe utilizando SDF, *code completion*, além de um interpretador para Stratego. Essas funcionalidades são distribuídas como um conjunto de *plugins* para a plataforma *Eclipse*. A ferramenta foi desenvolvida de maneira extensível, possibilitando que usuários escrevem programas Stratego que analisam o código escrito, a fim de verificar o estilo de código, por exemplo.

2.2.3 Rascal

Como já definido anteriormente, Rascal (3) é uma *DSL* proposta como uma tentativa de unificar ambos os domínios de análise e manipulação de código, uma vez que a maioria das feramentas existens até então trata desses problemas de maneira separada, especializando-se em algum dos dois domínios.

O domínio de *SCAM* utiliza uma série de conceitos, tais como gramáticas, *parsing*, ASTs, casamento de padrões, inferência de tipos, travessia de árvores, entre outros. Rascal propõe unificar esse conjunto de conceitos em uma única linguagem, com o objetivo de diminuir o esforço cognitivo e computacional necessário para integrar ferramentas existentes dentro do contexto de *SCAM*, além de fornecer um ambiente interativo para desenvolver aplicações.

Rascal foi projetado utilizando várias linguagens e ferramentas. Seus recursos relacionados à sintaxe são baseados na SDF (10), enquanto suas funcionalidades referentes à transformações e manipulações de programas são diretamente inspirados em linguagens baseadas em reescrita, como Stratego (2) ou ASF+SDF (12). Algumas ferramentas conhecidas pela comunidade também foram utilizadas como referência, tais como ANTLR (13) ou Eclipse IMP (14), pela sua facilidade de integração com um ambiente de desenvolvimento bem estabelecido.

Os principais requisitos que Rascal tenta atender são *expressividade*, *segurança* (no contexto de imutabilidade e sistema estático de tipos) e *usabilidade* (3). Algumas construções e recursos presentes na linguagem que atendem esses três requisitos são as seguintes:

- **Expressividade**: geração de arvóres sintáticas a partir de *parsers* que realizam o *parse* de um arquivo de código fonte a partir de uma definição formal de gramática, casamento de padrões, padrão de projeto *Visitor* utilizado na travessia de árvores sintáticas, utilização da sintaxe concreta na definição de regras de transformação.
- **Segurança**: sistema de tipos estático, imutabilidade, tratamento de exceções, ausência de conversões de tipo. Essas características previnem erros relacionados a tipo ou varíaveis não inicializadas, diminuindo as chances de ocorrerem erros em tempo de execução.
- **Usabilidade**: a linguagem Rascal é distribuída por meio de um *plugin* da plataforma Eclipse, uma plataforma já bem estabelecida e conhecida pela comunidade. Além disso, ela executa em cima da *Java Virtual Machine (JVM)*, outra plataforma estável que é usada há vários anos por usuários da linguagem Java. Por fim, sua sintaxe tenta se aproximar de linguagens de programação populares e bem conhecidas, como Java.

A Figura 2.3 mostra alguns recursos que atendem os requisitos propostos.

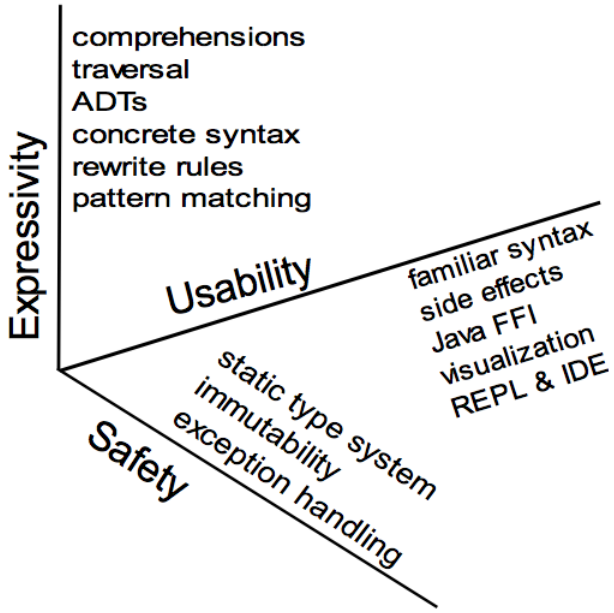


Figura 2.3: Requisitos atendidos pela linguagem Rascal (3).

O projeto final da linguagem Rascal possui uma estrutura em camadas, onde as camadas externas são construídas utilizando recursos das camadas mais internas. A Figura 2.4 ilustra essa estrutura.

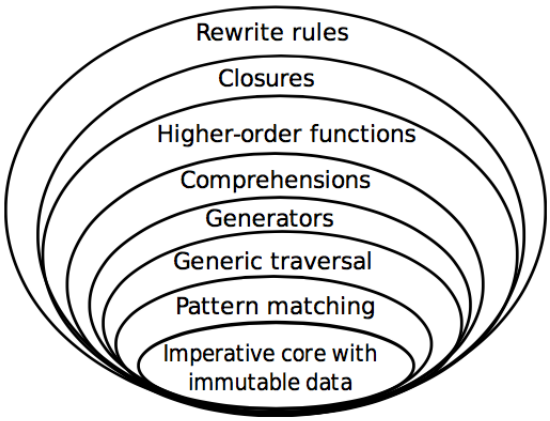


Figura 2.4: Estrutura em camadas da linguagem Rascal (3).

Rascal possui um rico conjunto de tipos de dados (3), variando de tipos mais básicos como *int* ou *string* até tipos mais complexos como listas, tuplas, árvores, além de tipos de dados algébricos, que podem ser utilizados para extender os tipos já definidos pela linguagem. A Tabela 2.1 mostra exemplos dos tipos de dados básicos definidos em Rascal.

Tabela 2.1: Tipos básicos presentes na linguagem Rascal

Tipo	**Exemplo**
bool	true, false
int	1, 0, -1, 123456789
real	1.0, 1.0232e20, -25.5
str	”abc”, ”first\nnext”
loc	file:///etc/passwd
tuple[$t_1, ..., t_n$]	$\langle 1, 2\rangle$, $\langle 1.0, 43, true\rangle$
list[t]	[], [1], [1,2,3], [true, 2, ”abc”]
set[t]	{}, {1,2,3,5,7}, {”john”,4.0}
map[t,u]	(), (1 : true,2 : true), (6 : {1,2,3,6},7 : {1, 7})
node	f, add(x, y), g(”abc”, [2, 3, 4])

A linguagem Rascal foi utilizada para a implementação das transformações deste trabalho por causa de sua facilidade de realizar transformações de uma maneira declarativa, abstraindo as dificuldades envolvidas no processo, devido aos recursos e construções citados acima. Em particular, a existência do padrão de projeto *Visitor* como construção de alto nível da linguagem e a possibilidade de realizar casamento de padrões utilizando elementos concretos da sintaxe da linguagem alvo da transformação facilitaram consideravelmente a escrita de transformações.

A listagem 2.1 apresenta um trecho de código Java antes e depois de ter sido transformado.

Listing 2.1: Exemplo de código Java.

```
// antes
if(1 == 1) {
    return true;
} else {
    return false;

}
// depois
return 1 == 1;
```

A listagem 2.2 apresenta o código Rascal que implementa essa transformação:

Listing 2.2: Exemplo de transformação Rascal.

```
CompilationUnit transformNaiveIfStatement(CompilationUnit unit) =
    visit(unit) {
        case (Statement) `if (<Expression cond>) { return true; } else {
            return false; }` => (Statement) `return <Expression cond>;`

};
```

No exemplo, ambos tipos *Statement* e *Expression* são elementos concretos da sintaxe da linguagem alvo da transformação (Java). É possível visualizar que a construção *visit* é utilizada para percorrer os nós de uma árvore de *parse* gerada a partir de um arquivo fonte. Além disso, é realizado o casamento de padrões em cima de um elemento concreto da sintaxe.

A listagem 2.3 apresenta um trecho de uma definição da gramática para a linguagem Java, que é utilizada para a construção de árvores de *parse*.

Listing 2.3: Exemplo de definição de gramática.

```
syntax Type = PrimitiveType
    | ReferenceType
    ;

syntax PrimitiveType = Annotation* NumericType
    | Annotation* "boolean"
    ;

syntax NumericType = IntegralType
    | FloatingPointType
    ;

syntax IntegralType = "byte"
    | "short"
    | "int"
    | "long"
    | "char"
    ;
```

Uma vez que as transformações foram construídas com o intuito de evoluir código Java para utilizar construções de versões mais novas da linguagem, a Seção a seguir descreve o histórico da linguagem Java, desde sua criação até os dias atuais.

2.3 Evolução da Linguagem Java

A linguagem Java era inicialmente chamada de *Oak*, e foi concebida pela *Sun Microsystems* no começo dos anos noventa com o intuito de ser utilizada em aplicações embarcadas, principalmente dispositivos interativos para uso pessoal. Com esse objetivo em vista, desde o princípio foi pensada como uma linguagem que poderia ser executada em vários ambientes, independente do hardware específico de cada dispositivo.

Por volta de 1995, a Sun voltou seus esforços para o uso da linguagem Java (já renomeada) em aplicações Web, na época uma tecnologia emergente que estava recebendo bastante atenção. A partir de 1999, na sua versão 2, a plataforma Java foi dividida em 3 especificações: J2SE (Java 2 Standard Edition), voltada para aplicações desktop, J2EE (Java 2 Enterprise Edition), voltada para aplicações Web, e J2ME (Java 2 Micro Edition), voltada para aplicações embarcadas em dispositivos eletrônicos.

Atualmente, a linguagem Java é amplamente difundida e utilizada uma variedade de domínios, sendo essencialmente uma linguagem Orientada a Objetos, com algumas construções típicas de linguagens funcionais adicionadas na sua versão 8. O problema inicial de ser uma linguagem que executa em qualquer dispositivo (*Write once, run anywhere*) foi solucionado por meio da *Java Virtual Machine*, ou JVM. A JVM é responsável por interpretar e executar um código intermediário denominado *bytecode*, que por sua vez é gerado a partir da compilação de um arquivo Java. Dessa forma, deve apenas existir uma implementação da JVM para cada dispositivo que deseje executar código Java, sem haver a necessidade de mudança do código original.

A Tabela 2.2 detalha as diferentes versões lançadas ao longo dos anos:

Tabela 2.2: Evolução dos recursos da linguagem Java

Versão	**Ano**	**Recursos**
1	1996	*Inner classes* (15), JDBC (16), internacionalização, suporte a Unicode
2	1998	Biblioteca *Collections* (17), API gráfica *Swing* (18)
3	2000	Implementação da JVM *HotSpot*
4	2002	Expressões regulares, suporte a IPv6, processamento de arquivos XML
5	2004	Generics API(19), anotações, enumerações, *autoboxing/unboxing*, *enhanced for each* (20)
6	2006	JDBC 4.0, melhoria nos algoritmos de *garbage collection*
7	2011	Operador diamante (21), *try with resources* (22), construção *switch-case* com variáveis do tipo *String*, *multi-catch* (23)
8	2014	Suporte a expressões lambda, interfaces funcionais e *default methods* (24), nova API de manipulação de datas
9	2017	Sistema de módulos, *java shell*
10	2018	Inferência de tipos para variáveis locais, coleção de lixo paralelizada.

As transformações que este trabalho visa evoluir têm como objetivo atualizar código de sistemas legados para utilizar construções introduzidas nas versões Java SE 7 e Java SE 8. Especificamente, as construções *multi-catch*, *Diamond operator* e o uso de expressões lambda em situações que utilizavam *Anonymous Inner Classes* (25) ou loops do tipo *enhanced for each*.

2.4 Trabalhos Relacionados

Este trabalho surgiu a partir de um estudo previamente realizado que apresenta a RJTL (6), uma biblioteca desenvolvida em Rascal para auxiliar a evolução de sistemas legados desenvolvidos em Java, visando transformar trechos de código legado em código que utilize construções introduzidas nas versões mais recentes da linguagem. A motivação para este estudo vem da necessidade de se manter uma base de código atualizada com novas construções de uma linguagem de programação (5).

Uma vez implementadas, as transformações foram aplicadas em um conjunto de projetos *open source*, a fim de verificar a percepção dos desenvolvedores destes projetos em relação à necessidade de se manter o código fonte aderente aos novos recursos e construções de uma linguagem de programação, bem como verificar quais transformações possuem uma maior taxa de aceitação e quais os motivos que podem levar a uma rejeição.

Durante os experimentos aplicando as transformações da RJTL nos projetos *open source*, viu-se necessária a evolução da implementação atual, especificamente nas verificações das pré-condições que viabilizam ou não a execução de uma certa transformação. Na implementação anterior, algumas pré-condições são ignoradas ou não são verificadas por completo, o que acarretou em alguns erros de compilação ou uma falha na suíte de testes de um projeto. Essa necessidade é a principal motivação para o presente trabalho.

As submissões para os projetos alvo foram feitas utilizando o mecanismo de *pull requests* da plataforma GitHub, onde os projetos eram hospedados. Foram submetidos 45 *pull requests* para 40 projetos diferentes, que continham 2462 transformações que alteraram 6399 linhas de código em 1207 arquivos de código fonte (6).

Os resultados foram classificados em três tipos: **aceitos**, **ignorados** e **rejeitados**, onde os rejeitados foram divididos em rejeições causadas por quebra de retrocompatibilidade com versões mais antigas do Java, ou pois os desenvolvedores dos projetos consideraram os resultados das transformações como inadequados. A Figura 2.5 ilustra os resultados obtidos.

A partir dos resultados, foi possível concluir que desenvolvedores de projetos *open source* aceitam contribuições para este tipo de refatoração quando a mudança no código tem um valor claro para os contribuidores. Um ponto de preocupação foi a existência de resultados em que a transformação não gerava uma melhoria no resultado final, o que foi o segundo motivo mais comum de rejeição das submissões, tendo ocorrido com mais frequência nas transformações que envolviam a introdução de expressões lambda.

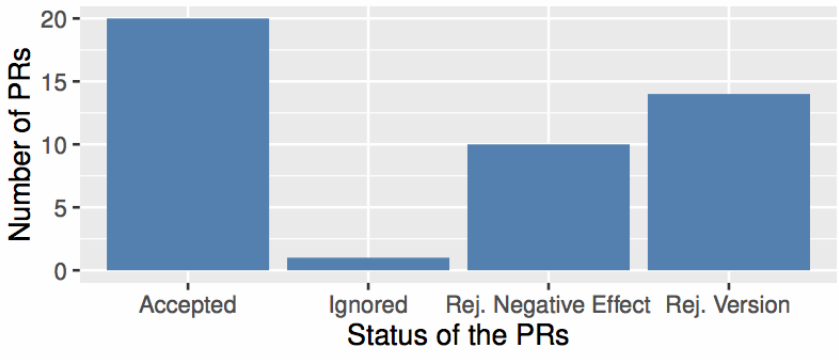


Figura 2.5: Resultados das submissões realizadas. (4).

Capítulo 3

Proposta de Solução

Nesse capítulo será descrita a proposta de solução desenvolvida ao longo do trabalho. Inicialmente, serão apresentadas as limitações encontradas na atual implementação da RJTL, a fim de justificar um dos problemas que a solução se propõe a resolver.

3.1 Limitações Atuais da RJTL

Durante a análise das transformações já implementadas previamente e dos resultados obtidos ao aplicar as transformações em projetos *open source*, foi possível observar que uma porção considerável dos falsos positivos e negativos se deram por causa da falta de verificação de pré-condições necessárias para que a transformação fosse realizada.

Neste contexto, um falso positivo consiste na aplicacão de uma transformação em um cenário que esta não é possível, devido ao não cumprimento de alguma das pré-condições. Um falso positivo tipicamente resulta em um erro de compilação no código transformado, e a transformação deve ser revertida manualmente antes de ser submetido o *pull request*. Por outro lado, um falso negativo consiste em um caso onde era possível aplicar uma transformação, mas a ferramenta não considerou este caso como elegível, devido a ter se focado em um conjunto mais restrito de cenários para aplicar as transformações (4).

Esses erros ocorreram principalmente nas transformações que envolviam o uso de expressões lambda, sendo elas a *AIC2Lambda*, que transforma uma *Anonymous Inner Class* para uma expressão lambda, e *Foreach2Lambda*, que transforma um loop do tipo *Enhanced For Loop* para uma expressão lambda. Além dessas duas, a transformação *Multicatch* também apresentou falsos positivos devido à falta de uma verificação semelhante. As figuras 3.1, 3.2 e 3.3 dão exemplos dessas três transformações em cenários passíveis de transformação:

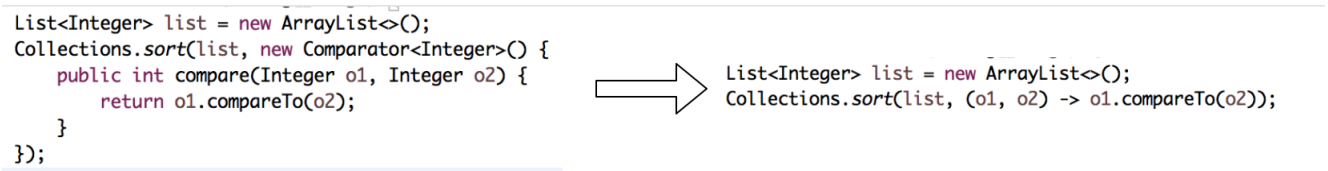


Figura 3.1: Exemplo de transformação AIC2Lambda.

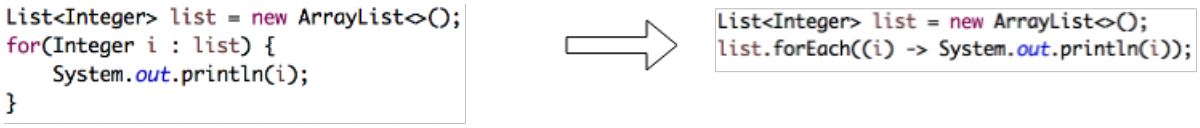


Figura 3.2: Exemplo de transformação Foreach2Lambda.

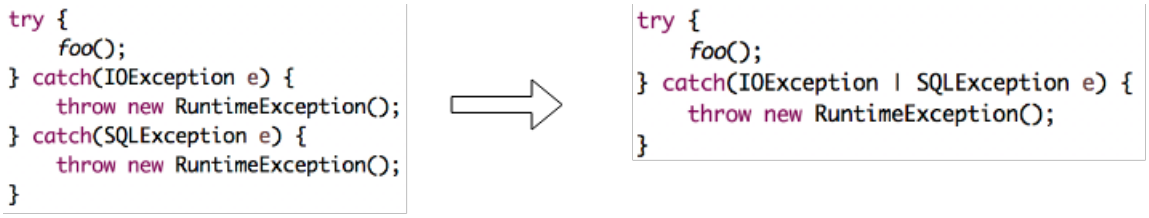


Figura 3.3: Exemplo de transformação MultiCatch.

A justificativa para a ocorrência dos falsos positivos e negativos é que essas transformações possuem pré-condições em que é necessário realizar verificações sobre os tipos envolvidos no código a ser transformado. Por questões de prazos e custos, a implementação atual da RJTL opta por deixar de lado essas verificações em alguns cenários ou por ser rigoroso demais em outros, acarretando nos já mencionados falsos positivos e negativos.

Em termos mais práticos, as transformações *AIC2Lambda* e *Foreach2Lambda* exigem que, no bloco de código que será transformado, não ocorra o lançamento de uma *checked exception* (26), ou seja, de uma exceção que não herde da classe *java.lang.RuntimeException*. Na implementação atual da transformação AIC2Lambda, essa verificação era feita de uma forma mais rigorosa, de maneira que caso existisse o lançamento de *qualquer* exceção, independente de ser *checked* ou *unchecked*, a transformação não ocorria. Isso resultou na ocorrência de falsos negativos, onde uma AIC cujo bloco de código lança uma exceção *unchecked* poderia ser transformado em uma expressão lambda, mas acabou sendo ignorado pela ferramenta. A listagem 3.1 mostra um trecho de código que não cumpre essa pré-condição, e a listagem 3.2 mostra um trecho que cumpre, mas que não foi identificado como elegível pela RJTL.

Listing 3.1: Trecho de código não elegível para a transformação AIC2Lambda.

```
Collections.sort(list, new Comparator<Integer>() {
    public int compare(Integer o1, Integer o2) {
      try {
         if(o1 < 0) throw new Exception();
      } catch (Exception e) {
         e.printStackTrace();
      }
      return o1.compareTo(o2);
    }
});
```

Listing 3.2: Trecho de código elegível que não foi identificado pela RJTL.

```
Collections.sort(list, new Comparator<Integer>() {
    public int compare(Integer o1, Integer o2) {
      if(o1 < 0) throw new IllegalArgumentException();
      return o1.compareTo(o2);
    }
});
```

A transformação *Foreach2Lambda* possui a mesma pré-condição em relação ao não lançamento de excecões do tipo *checked* (26), mas também possui uma pré-condição adicional, que é a de que a variável que está sendo iterada no loop precisa ser um subtipo da interface *java.util.Collection*. Sendo assim, os casos em que a variável iterada é um *Array* ou um subtipo da interface *java.util.Iterable* não podem ser transformados. Essa verificação possuia alguns erros na implementação atual e gerava falsos positivos. Especificamente, nos casos em que a variável iterada era um atributo da instância da classe e não de uma variável local ou argumento de um método, a pré-condição era ignorada. Os exemplos 4 e 4 mostram trechos de código em que a transformação é válida e inválida, respectivamente.

```
for(Integer i : new ArrayList<Integer>()) {
   System.out.println(i);
}
```

Listing 3.3: Trecho de código elegível para a transformação Foreach2Lambda.

```
for(Integer i : new Integer[10]) {
   System.out.println(i);
}
```

Listing 3.4: Trecho de código não elegível para a transformação Foreach2Lambda.

Por fim, a transformação *Multicatch* também exige uma pré-condição de que as classes envolvidas no bloco *multicatch* não podem possuir uma relação de subtipo. Essa pré-condição não era verificada na implementação atual da RJTL.

Com base nessa análise, foi possível concluir que era necessária uma maneira simples de se responder a pergunta: *Uma classe A é subtipo de uma classe B?*. Respondendo a essa pergunta, é possível verificar todas as pré-condições supracitadas, uma vez que para descobrir se uma exceção lançada é do tipo *checked* ou *unchecked*, basta verificar se a mesma é subtipo da classe *java.lang.RuntimeException*, direta ou indiretamente. A mesma pergunta resolve a pré-condição da transformação *Foreach2Lambda* em relação à variável que está sendo iterada no loop, bastando verificar se esta é um subtipo de *java.util.collection*. Por fim, para o caso da transformação *Multicatch*, uma verificação de subtipos entre as classes envolvidas no bloco *multicatch* pode ser respondida por essa pergunta.

Diante disso, a solução desenvolvida durante o trabalho se propõe a responder essa pergunta e auxiliar na verificação de pré -condições dessas transformações. Ainda existem outras pré-condições que não podem ser resolvidas por meio dessa análise de subtipos, mas essas não foram atacadas dentro deste trabalho.

Na seção a seguir é explicado o funcionamento do módulo construído para realizar a verificação de subtipos, utilizando a linguagem Java. Esse módulo foi denominado *Interface Rascal Java*.

3.2 Interface Rascal Java

A primeira tentativa de desenvolvimento de um módulo responsável por realizar análises relacionadas aos tipos e hierarquia entre as classes do projeto a ser transformado foi escrita em Rascal. Entretanto, foi possível perceber que a linguagem Rascal oferece um grande suporte em relação à análise sintática e transformações que envolvem apenas manipulação de nós de uma árvore sintática referente à uma unidade de compilação. Para uma análise relacionada aos tipos, que vai além de uma análise meramente sintática, algumas limitações do Rascal se evidenciaram e a tarefa se tornou bastante trabalhosa.

Com isso em mente, a opção seguinte foi utilizar Java para desenvolver esse módulo, uma vez que a integração entre Rascal e Java é dada de maneira simples, já que as duas linguagens são hospedadas pela JVM. A integração pode ser feita declarando um método dentro de um módulo Rascal com a palavra reservada *java* em sua assinatura, que será executado pelo código Java, e indicando o nome qualificado da classe Java a quem este método pertence. A listagem abaixo ilustra esse funcionamento.

```
// codigo Rascal
@javaClass{com.rascaljava.RascalJavaInterface}
java int initDB(str projectPath);

// codigo Java
package com.rascaljava;

import java.io.File;

public class RascalJavaInterface {
   public IValue initDB(IString projectPath) {
      return vf.integer(initDB(projectPath.getValue()));
   }
}
```

Listing 3.5: Exemplo de integração entre Rascal e Java.

3.2.1 Uso de BD em memória

A solução proposta para as verificações de tipos foi a de usar um Banco de Dados Relacional para armazenar informações sobre as classes do projeto a ser transformado. Para cada classe existente no código fonte do projeto, o BD armazena o nome qualificado da classe, a sua superclasse e interfaces que implementa, seus atributos e seus métodos. Para cada atributo e método, foi armazenado apenas seu nome e tipo (no caso do método, o seu tipo de retorno). Dessa maneira, antes de realizar alguma transformação, o BD seria populado com essas informações, podendo ser consultado durante da aplicação das transformações para responder as perguntas citadas na seção anterior.

Essa abordagem de se utilizar Bancos de Dados a fim de auxiliar na tarefa de refatoração já foi explorada anteriormente (27), particularmente utilizando um BD em memória ao invés de em disco, a fim de facilitar a utilização da RJTL, abordagem que também foi adotada neste trabalho. Dessa forma, o BD é criado em memória toda vez que a aplicação RJTL é iniciada, e os usuários não precisam se preocupar em configurar um BD externo para ser utilizado pela RJTL. O BD escolhido foi o H2DB, um BD em memória para Java.

A Figura 3.4 ilustra o Modelo Entidade Relacionamento projetado para o BD utilizado.

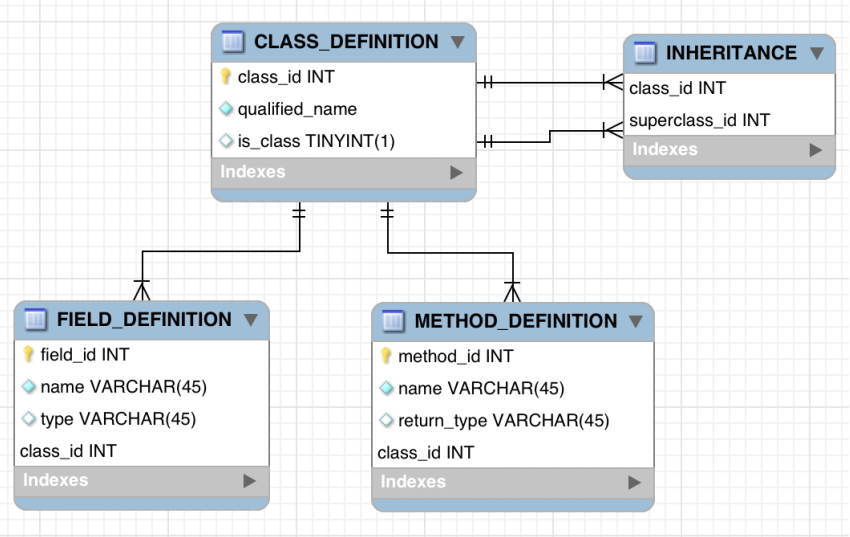


Figura 3.4: Modelo Entidade Relacionamento.

Como pode ser visto, existe uma tabela para cada uma das informações de classes, métodos e atributos, com um relacionamento 1-N entre a tabela *CLASS_ DEFINITION* e as tabelas *FIELD_ DEFINITION* e *METHOD_ DEFINITION*. A tabela *INHERITANCE* é uma tabela que representa a herança entre classes ou implementação de interfaces. Esse relacionamento é N-N, uma vez que uma classe pode implementar várias interfaces, e uma interface pode ser implementada por várias classes, portanto a tabela *INHERITANCE* é uma tabela associativa que guarda as chaves estrangeiras desse relacionamento N-N.

3.2.2 Uso da biblioteca Java Symbol Solver

Com o BD criado, o próximo passo é populá-lo com as informações das classes do projeto a ser transformado. Um problema que surge neste momento é que não é necessário populá-lo apenas com as informações das classes do projeto, mas também de classes pertencentes à quaisquer bibliotecas externas que o projeto utilize. Isso é necessário, por exemplo, nos casos em que é preciso verificar se uma exceção lançada é do tipo *checked* ou *unchecked*, uma vez que a exceção em questão pode ter sido declarada em uma biblioteca externa e não no código fonte do próprio projeto.

Para atacar este problema, fez-se uso das bibliotecas *Java Parser* e *Java Symbol Solver* (28), que realizam o *parse* de um arquivo fonte Java e possibilita a obtenção de informações sobre suas declarações de métodos e atributos, além de verificar possíveis superclasses e interfaces implementadas. Para este último caso, é necessário informar à biblioteca os

caminhos em que ela deve buscar "resolver" as interfaces e classes em questão. Esses caminhos podem ser diretórios locais, arquivos de biblioteca (com a extensão *jar*) ou a própria biblioteca padrão Java. A seguir, um exemplo de como isso funciona.

```
CombinedTypeSolver solver = new CombinedTypeSolver();
solver.add(new ReflectionTypeSolver());
solver.add(new JavaParserTypeSolver(new File("src/main/java")));
solver.add(new JarTypeSolver("lib/antlr-2.7.7.jar"));
```

Listing 3.6: Exemplo de uso do Java Symbol Solver.

Na listagem acima, é criado um objeto *solver* do tipo *CombinedTypeSolver*, uma classe definida pela biblioteca *Java Symbol Solver*. Em seguida, são adicionados os três tipos possíveis de caminhos onde o *solver* buscará informações de tipos que forem "resolvidos". O primeiro tipo é um *ReflectionTypeSolver*, que contém as classes da biblioteca padrão Java como *java.lang.** e *java.util.**. O segundo tipo, *JavaParserTypeSolver*, realiza a busca no diretório informado como argumento em sua inicialização, e o último, *JarTypeSolver*, realiza a busca em uma biblioteca *jar* informada.

Os dois primeiros tipos de *symbol solvers* podem ser utilizados sem maiores problemas, mas para o *JarTypeSolver*, foi necessário ter acesso a todos os arquivos *.jar* utilizados pelo projeto a ser transformado. A maioria dos projetos *open source* que foram analisados faz uso de algum gerenciador de dependências, como *Maven* ou *Gradle*, o que faz com que as bibliotecas *jar* não estejam presentes explicitamente dentro do projeto, sendo na verdade gerenciadas por essas ferramentas em algum diretório externo.

Para obter acesso a esses arquivos *jar*, foi utilizado um recurso do gerenciador de dependências *Maven* que realiza o *download* de todas as dependências do projeto em uma única pasta de destino. Esse recurso só é possível para projetos que utilizam o *Maven* como gerenciador, portanto foi necessário restringir o escopo de projetos a serem analisados para conter apenas projetos *Maven*. Dessa forma, foi possível montar o caminho de busca para um *CombinedTypeSolver* que contém todos os arquivos fonte do projeto, bem como suas dependências e as classes da biblioteca padrão Java.

O processo final de população do BD consiste então em varrer os arquivos fonte do projeto a ser transformado antes da aplicação das transformações e persistir as informações necessárias, utilizando o recurso da biblioteca *Java Symbol Solver* para encontrar informações relacionadas à hierarquia de classes e interfaces. A inicialização do *CombinedTypeSolver* é realizada da seguinte maneira:

```
public CombinedTypeSolver initTypeSolver(String projectPath) {
      solver = new CombinedTypeSolver();
      solver.add(new JavaParserTypeSolver(new File(projectPath)));
      solver.add(new ReflectionTypeSolver());
      String rootPath = copyProjectJars(projectPath);
      List<File> jars = IOUtil.findAllFiles(rootPath + "/dependencies", "jar");
      jars.forEach((jar) -> {
         try {
            solver.add(new JarTypeSolver(jar.getAbsolutePath()));
         } catch (IOException e) {
            e.printStackTrace();
         }
```

```
    });
    return solver;}
```

3.2.3 Verificação de Pré-Condições

Uma vez populado o BD com as informações necessárias, as transformações podem ser aplicadas com o auxílio da Interface Rascal Java. Para isso, foram criados dois métodos Java, descritos abaixo:

```
@javaClass{com.rascaljava.RascalJavaInterface}
java bool isRelated(str clazzA, str clazzB);

@javaClass{com.rascaljava.RascalJavaInterface}
java bool isCollection(str className, str fieldName);
```

O primeiro método verifica se uma classe A é subclasse de uma classe B ou vice-versa, e o segundo verifica se um determinado atributo de uma classe implementa a interface *java.util.Collection*. Como já explicado anteriormente, essas duas verificações são relevantes para as transformações *AIC2Lambda*, *Foreach2Lambda* e *Multicatch*.

O método *isRelated* consulta o BD para cada uma das duas classes passadas como argumento, a fim de montar uma lista, com todas as classes e interfaces que possuem alguma relação com as classes em questão. Esse procedimento é realizado por meio de uma busca recursiva no BD, visitando as superclasses e interfaces, que estão representadas como entradas na tabela *INHERITANCE*, até que seja encontrada a classe *java.lang.Object*, que não possui ancestrais e representa o caso base da recursão. O código abaixo detalha esse algoritmo:

```
public List<ClassDefinition> getAllAncestors (String qualifiedName) {
    StringBuilder sb = new StringBuilder();
   sb.append("SELECT a.* from CLASS_DEFINITION a ");
   sb.append("INNER JOIN INHERITANCE b on a.class_id = b.superclass_id ");
   sb.append("INNER JOIN CLASS_DEFINITION c on c.class_id = b.class_id ");
   sb.append("WHERE c.qualified_name = ?");
   PreparedStatement stmt = connection.prepareStatement(sb.toString());
   stmt.setString(1, qualifiedName);
   ResultSet rs = stmt.executeQuery();
   List<ClassDefinition> ancestors = new ArrayList<>();
   List<ClassDefinition> list = new ArrayList<>();
   boolean goOn = true;
   while(rs.next()) {
      ancestors.add(buildClassDefinition(rs));
      if(classDef.getQualifiedName().equals("java.lang.Object")) {
         goOn = false;
      }
   }
   if(goOn) {
       for(ClassDefinition ancestor : list) {
          ancestors.addAll(getAllAncestors(ancestor.getQualifiedName()));
```

```
      }
      return ancestors;
   } else { return ancestors;}
}
```

Uma vez montadas essas duas listas de ancestrais das classes A e B, basta uma simples checagem se a classe B está contida na lista de ancestrais de A e vice-versa. Portanto, o código final do método *isRelated* se dá na seguinte forma:

```
public boolean isRelated(String clazzA, String clazzB) {
   String qualifiedClazzA = DB.getInstance().findQualifiedName(clazzA);
   String qualifiedClazzB = DB.getInstance().findQualifiedName(clazzB);
   if(qualifiedClazzA.equals(qualifiedClazzB)) {
      return true;
   }
   List<ClassDefinition> classAAncestors =
       DB.getInstance().getAllAncestors(qualifiedClazzA);
   List<ClassDefinition> classBAncestors =
       DB.getInstance().getAllAncestors(qualifiedClazzB);
   return classAAncestors.stream().anyMatch((a) ->
       a.getQualifiedName().equals(qualifiedClazzB)) ||
         classBAncestors.stream().anyMatch((a) ->
             a.getQualifiedName().equals(qualifiedClazzA));
}
```

Na listagem abaixo é exemplificado o uso da verificação *isRelated* na transformação *Multicatch*, verificando se uma determinada exceção herda de alguma outra exceção presente no bloco *try-catch*.

```
private bool areExceptionsRelated(CompilationUnit unit, list[CatchType] types,
    CatchType t) {
   bool related = false;
   list[str] typesStr = [ trim(unparse(t)) | t <- types ];
   list[str] qualifiedTypes = getQualifiedTypes(unit);
   for(str tp <- qualifiedTypes) {
      if(isRelated(tp, t)) { // metodo da RascalJavaInterface
         related = true;
         break;
      }
   }

    return related;
}
```

A implementação do método *isCollection* é relativamente simples, bastando apenas recuperar o tipo do atributo *fieldName* informado como parâmetro, e fazer uma chamada ao método *isRelated*, passando esse tipo e a interface *java.util.Collection* como argumentos, concluindo assim se o tipo "é uma" *Collection* ou não. Um exemplo de sua utilização pode ser visto na listagem abaixo, extraída da transformação *ForEach2Lambda*.

```
private bool isIdentifierACollection(CompilationUnit unit, Expression exp,
    set[MethodVar] availableVariables) {
   varName = trim(unparse(exp));
   varName = replaceFirst(varName, "this.", "");
   className = findCurrentClassName(unit);
   var = findByName(availableVariables, varName);
   if(var.isClassField) {
      return isCollection(className, var.name); // metodo da RascalJavaInterface
   } else {
      return !isTypePlainArray(var) && !isIterable(var);
   }
}
```

3.2.4 Visão geral da Interface Rascal Java

Para concluir a explicação sobre o código construído, a seguir será ilustrado o funcionamento geral da RJTL com a adição da Interface Rascal Java. Primeiramente, a Figura 3.5 mostra a organização do módulo Java desenvolvido (alguns métodos internos foram omitidos):

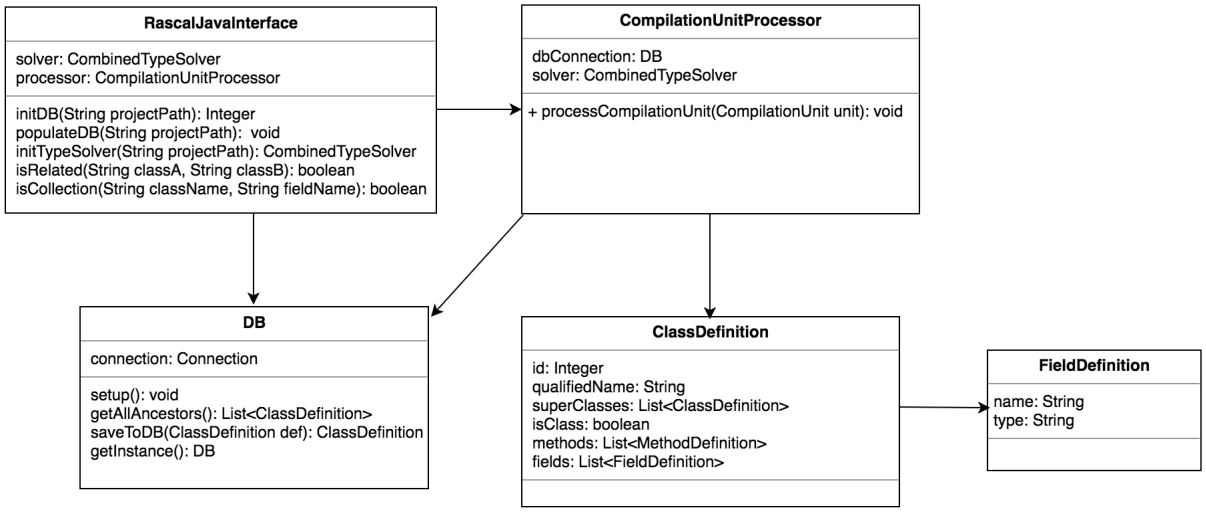


Figura 3.5: Organização da Interface Rascal Java.

O ponto de entrada é a classe *RascalJavaInterface*, através dos métodos *initDB*, *isRelated* e *isCollection*, que são chamados pelo módulo Rascal. O fluxo de execução final, incluindo os passos que envolvem o módulo Rascal, é ilustrado pela Figura 3.6.

Isso encerra a explicação do funcionamento da Interface Rascal Java, e como ela resolve o problema da verificação de pré-condições, que era uma das maneiras de evoluir a implementação atual da RJTL, objetivo deste trabalho. As outras duas maneiras propostas serão explicadas a seguir.

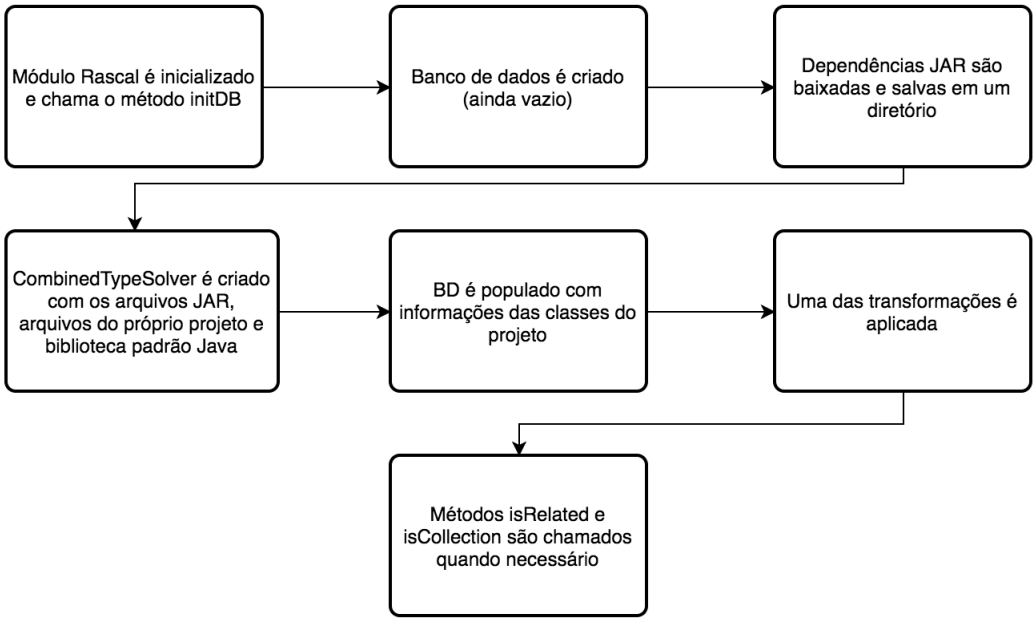


Figura 3.6: Fluxograma de execução da RJTL.

3.3 Evolução da Transformação Diamond Operator

Uma outra evolução identificada na implementação atual da RJTL foi na transformação *Diamond Operator*. Essa transformação introduz o operador diamante (21), presente na linguagem Java desde sua versão 7. A transformação detecta inicializações que utilizam tipos genéricos e os remove no lado direito da inicialização, como exemplificado na Figura 3.7:

```
List<Integer> list = new ArrayList<Integer>();  ⟹  List<Integer> list = new ArrayList<>();
```

Figura 3.7: Exemplo da transformação Diamond Operator.

A limitacão atual encontrada foi a de que a transformação é aplicada apenas no caso em que ocorre a declaração e inicialização de uma varíavel ou atributo no mesmo momento. Isso quer dizer que o cenário em que uma varíavel é apenas declarada em um primeiro momento e inicializada em um momento posterior não é identificado pela RJTL. A Figura 3.8 ilustra um exemplo de como funcionaria essa transformação, que atualmente não é detectada:

```
List<Integer> list;                          List<Integer> list;
list = new ArrayList<Integer>();     ⟹      list = new ArrayList<>();
```

Figura 3.8: Exemplo de transformação que não é detectada atualmente pela RJTL.

Dessa forma, nesse trabalho foi proposta a evolução da transformação *Diamond Operator* para também identificar os cenários desse segundo exemplo. A implementação desse

novo cenário foi simples e não envolveu a Interface Rascal Java, uma vez que não é necessário realizar nenhuma verificação de tipos, bastando este novo caso em que ocorre uma inicializacão de uma varíavel que possui tipos genéricos após ter sido declarada. Vale ressaltar que este cenário é aplicável apenas em variáveis locais e não atributos de classe, uma vez que nestes é necessária a inicialização no momento da declaração.

3.4 Transformação Method Reference

A última proposta de evolução da implementação atual da RJTL foi a criação de uma nova transformação, denominada *Method Reference*. Essa transformação introduz o uso do recurso *Method Reference* (29), presente no Java desde sua versão 8. A ideia de sua utilização é substituir o uso de expressões lambda que simplesmente chamam um método de um objeto pela referência a este método, tornando o código menos verboso. O recurso *Method Reference* também pode ser utilizado em outros contextos como a referência a um método estático ou ao construtor de uma classe, que não foram implementados nesse trabalho.

A Figura 3.9 dá um exemplo dessa transformação:

```
List<String> list2 = new ArrayList<>();
list2.stream().map((s) -> s.length());
```

⟹

```
List<String> list2 = new ArrayList<>();
list2.stream().map(String::length);
```

Figura 3.9: Exemplo da transformação Method Reference.

É importante ressaltar que esta transformação foge um pouco do escopo inicial da RJTL, que era implementar transformações que substituem construções antigas de projetos legados por novos recursos introduzidos em versões mais atuais da linguagem Java, uma vez que ambos os recursos foram adicionados ao Java na sua versão 8. No entanto, é uma transformação simples de ser implementada e que traz uma melhoria clara no código, por isso foi tomada a decisão de implementá-la.

Assim como na evolução da transformação *Diamond Operator*, não foi necessária a utilização da Interface Rascal Java. A implementação atual do *Method Reference* se limitou a aplicar a transformação apenas em cenários onde a variável que utiliza a expressão lambda é uma variável local ou argumento de método, excluindo os casos em que é um atributo de classe.

A seguir será explicado o funcionamento da transformação. Inicialmente, dentro do contexto de um método de uma determinada classe, é necessário fazer uma varredura nas inicializações de varíavel e armazenar seus tipos, já que a construção *Method Reference* exige que seja informado o tipo e o método que será executado. A Figura 3.10 demonstra essa varredura:

```
case(LocalVariableDeclaration)
    `<VariableModifier* modif> <UnannType varType> <VariableDeclaratorId varName> = <VariableInitializer init>`:
    if(/\<<lType:.*>\>/ := unparse(varType)) {
        listTypes[trim(unparse(varName))] = trim(lType);
    }
}
```

Figura 3.10: Armazenamento de tipos para variáveis locais.

A expressão *case* restringe a busca aos nós da árvore sintática em questão àqueles que são do tipo *LocalVariableDeclaration*, que corresponde a um elemento de sintaxe da gramática definida para a linguagem Java. Para capturar o tipo da variável foi utilizada uma expressão regular, filtrando apenas os casos em que o tipo da variável contém um tipo genérico, informado entre os caracteres <>, e armazenando esse tipo genérico. Essa restrição para tipos genéricos se mostrou adequada uma vez que um caso de uso comum para expressões lambda é durante o processamento de listas, que são implementadas utilizando tipos genéricos.

Com os tipos devidamente armazenados, o próximo passo é encontrar chamadas de métodos dentro do método que está sendo analisado, a fim de encontrar a declaração de uma expressão lambda. Essa expressão lambda deve conter apenas uma única expressão em seu corpo para que seja transformada em um *Method Reference*. Essas verificações são realizadas utilizando o recurso de casamento de padrões presente no Rascal, fazendo com que cenários que não são apropriados para transformação não sejam "casados", sendo assim ignorados pela transformação. A Figura 3.11 ilustra esse funcionamento :

```
case(MethodInvocation)`<ExpressionName listName>.<Identifier methodName>(<ArgumentList? methodArgs>)` : {
    listNameStr = trim(unparse(listName));
    refactoredArgs = visit(methodArgs) {
        case(Expression)
            `(<Identifier id>) -\> <ExpressionName name>.<TypeArguments? tArgs> <Identifier mName>()` : {
            <parsed, methodReference> = parseMethodReference(trim(unparse(id)), trim(unparse(name)),
            trim(unparse(methodName)), listNameStr, listTypes);
            if(parsed) {
                total += 1;
                insert methodReference;
            }
        }
```

Figura 3.11: Exemplo de casamento de padrões na transformação Method Reference.

Por fim, o último passo é substituir o corpo da expressão lambda por uma construção *MethodReference*. Isso é feito pela função *parseMethodReference*, que simplesmente constrói um novo elemento sintático que corresponde ao *Method Reference* e substitui o nó correspondente ao corpo da expressão lambda por essa nova expressão. Essa construção é ilustrada pela Figura 3.12:

```
if(listName in listTypes && lambdaArgId == lambdaVarId) {
    methodRef = parse(#MethodReference, "<listTypes[listName]>::<methodName>");
    return <true, parse(#Expression, unparse(methodRef))>;
}
```

Figura 3.12: Construção da expressão Method Reference.

Com isso, se encerra a explicação das evoluções da biblioteca RJTL. Na seção a seguir são descritos os resultados encontrados ao aplicar a RJTL, contendo essas novas implementações, em alguns projetos Java.

3.5 Avaliação dos Resultados

Para avaliar as evoluções implementadas, optou-se por aplicar as transformações em um conjunto de projetos, alguns deles já tendo sido utilizados no trabalho inicial de construção da RJTL e outros não.

No caso da Interface Rascal Java, a avaliação consistiu em verificar se houve uma queda no número de falsos positivos e negativos após a sua construção. Para isso, as transformações da RJTL foram aplicadas em dois momentos, o primeiro sem as evoluções implementadas e o segundo com as mesmas adicionadas à RJTL. Feito isso, foi feita uma comparação para verificar se houve alguma queda no número de falsos positivos e negativos.

Para a transformação *Diamond Operator* e *Method Reference*, a avaliação consistiu em identificar as transformações realizadas pela RJTL com a adição das duas implementações nos projetos escolhidos.

3.5.1 Resultados da Interface Rascal Java

As transformações aplicadas para se avaliar os resultados da Interface Rascal Java foram a *Multicatch* e *AIC2Lambda*. Os projetos analisaodos foram: *Elasticsearch*, *Rascal*, *JavaParser*, *OrientDB*, *Apache Storm*, *Spring Boot*, *Sisdot* e *Sisbol*.

Foi encontrada uma queda no número de falsos positivos na aplicação da transformação *MultiCatch* no projeto *OrientDB*. Na implementação inicial da RJTL, foram realizadas 8 transformações, das quais 5 delas geraram um erro de compilação devido à quebra da pré-condição citada previamente. Ao aplicar a transformação com a verificação realizada pela Interface Rascal Java, as 5 transformações não foram aplicadas, tendo assim eliminado 5 casos de falsos positivos.

3.5.2 Resultados da Transformação Diamond Operator

Para avaliar a evolução na transformação *Diamond Operator* foi aplicada a transformação nos mesmos projetos já citados acima. Seus resultados são apresentados abaixo:

Tabela 3.1: Resultados na transformação Diamond Operator

Projeto	**Transformações realizadas**
Elasticsearch	10
Rascal	6
JavaParser	22
OrientDB	226
Storm	303
Spring Boot	0
Sisdot	1
Sisbol	3

A seguir, alguns exemplos de transformações realizadas:

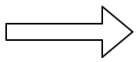
```
supervisors = new HashSet<String>();
admins = new HashSet<String>();
```

```
supervisors = new HashSet<>();
admins = new HashSet<>();
```

Figura 3.13: Exemplo de Diamond Operator no projeto Apache Storm.

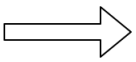
```
if (seconds == null) {
    seconds = new TreeSet<Integer>();
}
if (minutes == null) {
    minutes = new TreeSet<Integer>();
}
```

```
if (seconds == null) {
    seconds = new TreeSet<>();
}
if (minutes == null) {
    minutes = new TreeSet<>();
}
```

Figura 3.14: Exemplo de Diamond Operator no projeto Elasticsearch.

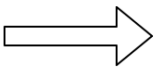
```
if (linkOffsets == null) {
    linkOffsets = new ArrayList<Integer>();
    linkLengths = new ArrayList<Integer>();
    linkTargets = new ArrayList<String>();
}
```

```
if (linkOffsets == null) {
    linkOffsets = new ArrayList<>();
    linkLengths = new ArrayList<>();
    linkTargets = new ArrayList<>();
}
```

Figura 3.15: Exemplo de Diamond Operator no projeto Rascal.

3.5.3 Resultados da Transformação Method Reference

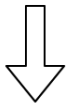
O processo utilizado para a avaliação da transformação *Method Reference* foi o mesmo utilizado na transformação *Diamond Operator*. Os seguintes resultados foram encontrados:

Tabela 3.2: Resultados da transformação Method Reference

Projeto	**Transformações realizadas**
Elasticsearch	9
Rascal	1
JavaParser	3
OrientDB	24
Storm	6
Spring Boot	2
Sisdot	0
Sisbol	0

A seguir, alguns exemplos de transformações realizadas:

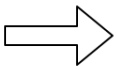
```
endpointBeans.stream()
.collect(Collectors.toMap((bean)->bean.getType(), (bean)->bean));
```



```
endpointBeans.stream()
.collect(Collectors.toMap(EndpointBean::getType, (bean)->bean));
```

Figura 3.16: Exemplo de Method Reference no projeto Spring Boot.

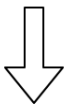
```
Set<Long> ids = threads
        .stream()
        .map(t -> t.getId())
        .collect(Collectors.toSet());
```



```
Set<Long> ids = threads
        .stream()
        .map(Thread::getId)
        .collect(Collectors.toSet());
```

Figura 3.17: Exemplo de Method Reference no projeto Elasticsearch.

```
Set<TopicPartition> assignedTps = partitionResponsibilities
                                  .stream()
                                  .map(kttp -> kttp.getTopicPartition())
                                  .collect(Collectors.toSet());
```



```
Set<TopicPartition> assignedTps = partitionResponsibilities
                                  .stream()
                                  .map(KafkaTridentSpoutTopicPartition::getTopicPartition)
                                  .collect(Collectors.toSet());
```

Figura 3.18: Exemplo de Method Reference no projeto Apache Storm.

Esses resultados corroboram com a análise realizada na implementação inicial da RJTL de que a linguagem Rascal é bastante eficaz para transformações que dependem apenas da análise sintática do código, como é o caso de ambas as transformações *Diamond Operator* e *Method Reference*.

No próximo, e último, capítulo deste trabalho será discutida a conclusão do estudo realizado e possíveis trabalhos futuros.

Capítulo 4

Conclusão

Neste trabalho foram propostas maneiras de se evoluir a biblioteca RJTL, uma biblioteca de transformações de programas Java desenvolvida em Rascal. Para isso, foram atacadas algumas das limitações e erros presentes nas transformações previamente implementadas, por meio do desenvolvimento de um módulo Java, denominado Interface Rascal Java, responsável por realizar algumas verificações referentes aos tipos e a hierarquia das classes envolvidas nas transformações.

Além da Interface Rascal Java, também foi implementada uma nova transformação que traz uma melhoria na legibilidade do código, diferentemente das transformações previamente implementadas, que visavam atualizar projetos legados para utilizar novas construções da linguagem Java.

Para medir os resultados dessas implementações, as transformações impactadas pelas mudanças foram aplicadas em projetos *open source*, a fim de identificar a efetividade das evoluções. Os resultados obtidos em relação à nova transformação implementada (*Method Reference*) e à evolução na transformação *Diamond Operator* foram positivos, apresentando várias oportunidades de refatoração. No entanto, a Interface Rascal Java não trouxe ganhos satisfatórios na aplicação das transformações, apesar de ter consumido o maior esforço de implementação dentre todas as evoluções.

Apesar disso, a construção da Interface Rascal Java pode servir para possíveis trabalhos futuros, já que as informações armazenadas podem ser utilizadas para realizar outros tipos de verificações. Quaisquer transformações que precisem realizar verificações de tipos sobre métodos ou atributos de uma determinada classe, por exemplo, podem se beneficiar dessa infraestrutura existente.

Durante a realização deste trabalho, algumas possibilidades de trabalhos futuros foram identificadas. O principal ponto identificado é a reescrita da transformação *Foreach2Lambda*, cuja implementação se tornou bastante complexa e pode ser reescrita de maneira mais simples utilizando a Interface Rascal Java. Também foram discutidas situações em que esta transformação pode ser utilizada de maneiras diferentes para poderem trazer melhorias mais significativas ao código, como o uso de expressões *map* ou *filter*, que são comuns no paradigma de Programação Funcional.

Também é possível evoluir a transformação *Method Reference*, desenvolvida neste trabalho, para abarcar outros cenários em que é possível utilizar essa construção, como em expressões lambda que fazem uma chamada a métodos estáticos ou a construtores. Du-

rante o trabalho foi considerado apenas um dos cenários possíveis, já que não era sabida a aplicabilidade dessa transformação.

Um outro ponto de evolução é expandir a RJTL para aplicar refatorações em outras linguagens, algo que é factível sem muita dificuldade por meio do uso da linguagem Rascal e que ainda não está sendo muito explorado por trabalhos recentes da área. Particularmente, a linguagem *JavaScript* tem passado por várias mudançås nos últimos anos, muitas delas podendo ser atualizadas de forma automatizada por meio de uma ferramenta como a RJTL.

Referências

[1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. Stratego/xt 0.17. a language and toolset for program transformation. *Sci. Comput. Program.*, 72(1-2):52–70, June 2008. ix, 5

[2] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In Aart Middeldorp, editor, *Rewriting Techniques and Applications*, pages 357–361, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ix, 5, 6, 7

[3] P. Klint, T. v. d. Storm, and J. Vinju. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*, pages 168–177, Sept 2009. ix, 1, 5, 7, 8, 9

[4] Reno Medeiros Dantas. Transformacao de Programas para Suportar a Evolucao da Linguagem Java. Master's thesis, Universidade de Brasília, Departamento de Ciência da Computação, 2017. ix, 13, 14

[5] Jeffrey Overbey and Ralph E. Johnson. Regrowing a language: refactoring tools allow programming languages to evolve. In *OOPSLA*, 2009. 1, 12

[6] R. Dantas, A. Carvalho, D. Marcílio, L. Fantin, U. Silva, W. Lucas, and R. Bonifácio. Reconciling the past and the present: An empirical study on the application of source code transformations to automatically rejuvenate java programs. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 497–501, March 2018. 1, 12

[7] Martin Fowler. *Refactoring: improving the design of existing code.* Addison-Wesley Professional, 1999. 4

[8] Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, pages 109–143, 2001. 4

[9] Eelco Visser. Meta-programming with concrete object syntax. In *Proceedings of the 1st ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering*, GPCE '02, pages 299–315, London, UK, UK, 2002. Springer-Verlag. 6

[10] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers. The syntax definition formalism sdf - reference manual -, 1992. 6, 7

[11] Karl Trygve Kalleberg and Eelco Visser. Spoofax: An interactive development environment for program transformation with Stratego/XT. In A. Sloane and A. Johnstone, editors, *Seventh Workshop on Language Descriptions, Tools, and Applications (LDTA'07)*, pages 47–50, Braga, Portugal, March 2007. 7

[12] M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment: A component-based language development environment. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 365–370, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. 7

[13] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. Pragmatic Bookshelf, 2007. 7

[14] Philippe Charles, Robert M. Fuhrer, and Stanley M. Sutton. Imp: a meta-tooling platform for creating language-specific ides in eclipse. In *ASE*, 2007. 7

[15] Oracle. Nested classes. Disponível em: `https://docs.oracle.com/javase/tutorial/java/javaOO/nested.html`. Acessado em: 06/05/2018. 11

[16] Oracle. Jdbc. Disponível em: `http://www.oracle.com/technetwork/java/javase/jdbc/index.html`. Acessado em: 06/05/2018. 11

[17] Oracle. The collections framework. Disponível em: `https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html`. Acessado em: 06/05/2018. 11

[18] Oracle. Swing api. Disponível em: `https://docs.oracle.com/javase/8/docs/technotes/guides/swing/index.html`. Acessado em: 06/05/2018. 11

[19] Oracle. Generics. Disponível em: `https://docs.oracle.com/javase/tutorial/java/generics/index.html`. Acessado em: 06/05/2018. 11

[20] Oracle. The for-each loop. Disponível em: `https://docs.oracle.com/javase/8/docs/technotes/guides/language/foreach.html`. Acessado em: 06/05/2018. 11

[21] Oracle. Type inference for generic instance creation. Disponível em: `https://docs.oracle.com/javase/7/docs/technotes/guides/language/type-inference-generic-instance-creation.html`. Acessado em: 06/05/2018. 11, 23

[22] Oracle. The try with resources statement. Disponível em: `https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html`. Acessado em: 06/05/2018. 11

[23] Oracle. Catching multiple exception types. Disponível em: `https://docs.oracle.com/javase/8/docs/technotes/guides/language/catch-multiple.html`. Acessado em: 06/05/2018. 11

[24] Oracle. Functional interfaces. Disponível em: `https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html`. Acessado em: 06/05/2018. 11

[25] Oracle. Anonymous classes. Disponível em: `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`. Acessado em: 06/05/2018. 12

[26] Alex Gyori, Lyle Franklin, Danny Dig, and Jan Lahoda. Crossing the gap from imperative to functional programming through refactoring. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 543–553, New York, NY, USA, 2013. ACM. 15, 16

[27] J. Kim, D. Batory, D. Dig, and M. Azanza. Improving refactoring speed by 10x. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 1145–1156, May 2016. 18

[28] Java parser. Disponível em: `https://github.com/javaparser/javaparser`. Acessado em: 01/06/2018. 18

[29] Oracle. Method reference. Disponível em: `https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html`. Acessado em: 06/05/2018. 24